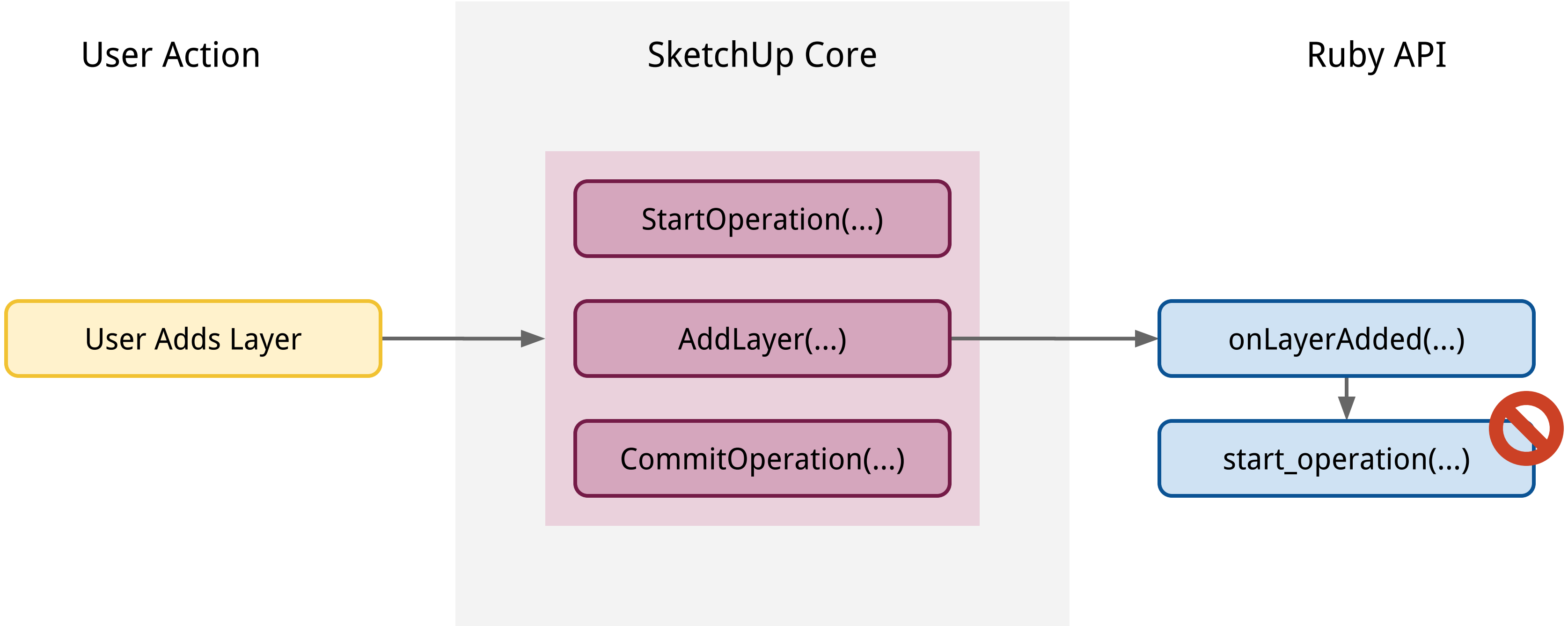
Observers & Operations

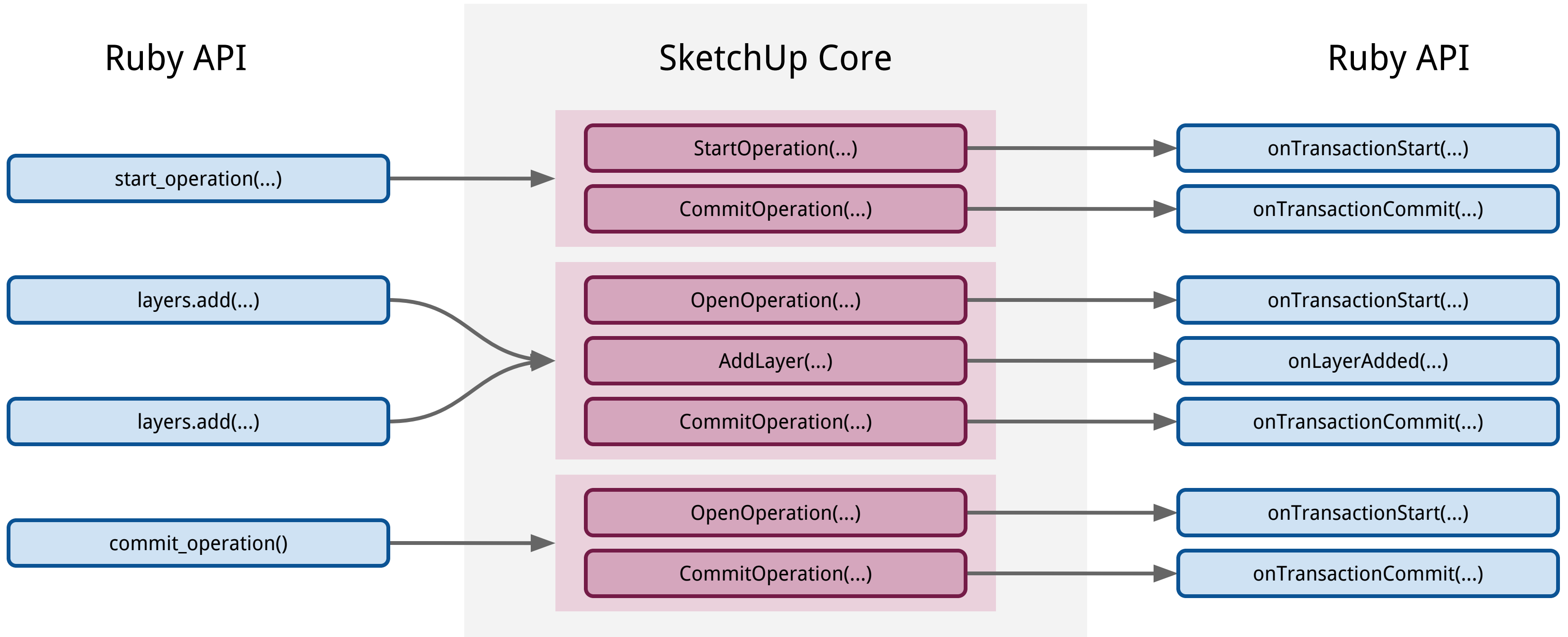
SketchUp 2016 Changes and Best Practices

Previous Observer Behaviour



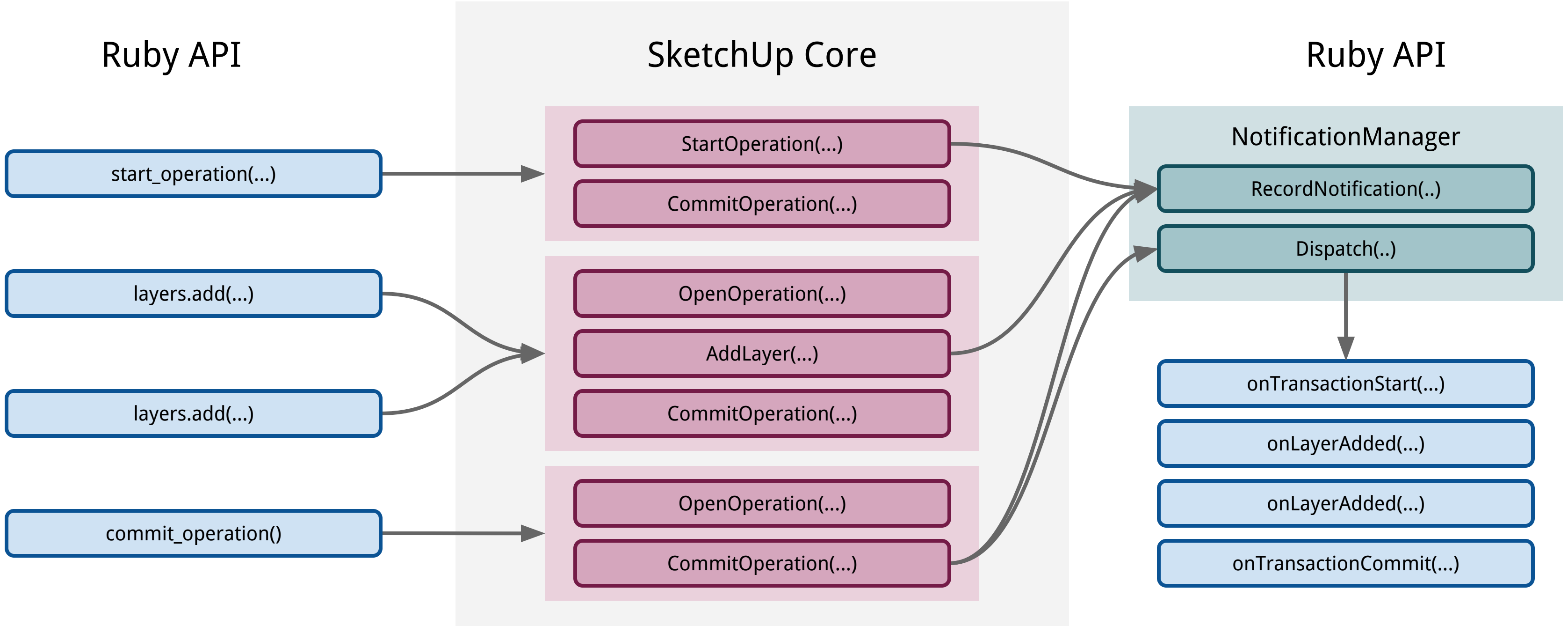
Problem: Observer is triggered immediately while the operation is still open. This means the observer could disrupt the undo stack by starting a new operation while one is already active. A second problem would be if the observer removed entities the current operation is about to use.

Previous Observer Behaviour



Problem: The Ruby API creates intermediate operations and the ModelObserver doesn't filter them out. This makes it impossible to know when a Ruby operation has truly ended. This is why we had to resort to the ugly workaround of using a timer to delay model changes: github.com

SU2016 Observer Behaviour



Change: Ruby API observer events are queued up until the active operation is done. Intermediate Ruby operations doesn't trigger the ModelObserver's onTransaction* events.

SU2016 Observer Behaviour

Ruby observers are not triggered until the current operation has completed.

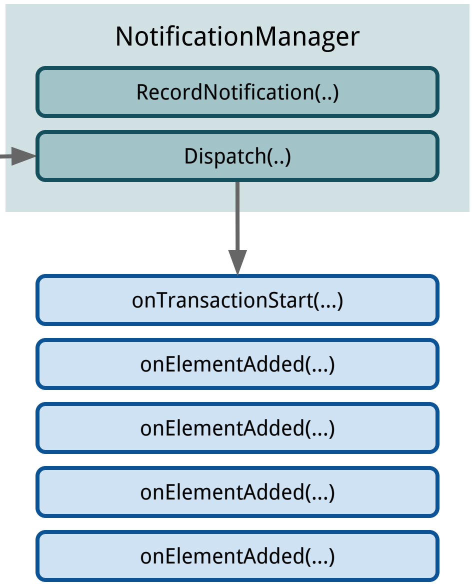
...except...

```
cmd = UI::Command.new('Hello World') {
  Example.operation_bad_example
  # Even if this command started an operation which wasn't closed because
  # it was interrupted by an error, any notifications queued up will be
  # dispatched after SketchUp trigger this proc.
}
```

```
module Example

  def self.operation_bad_example
    model = Sketchup.active_model
    entities = model.active_entities
    model.start_operation('Hello World', true)
    5.downto(0) { |i|
      z = 5 / i # Will eventually raise ZeroDivisionError
      face = entities.add_face([0, 0, z], [0, 9, z], [9, 9, z], [9, 0, z])
      face.set_attribute('Example', 'Hello', "World #{i}")
    }
    # This will not be reached when the error is raised - leaving the Ruby
    # operation open!
    model.commit_operation
  end

end # module
```



Notifications are still sent out after a user interaction even if a Ruby operation is left open. This is to ensure that extensions such as render engines can update their live render even though an extension failed to close its operation.

Such user interactions include:

- Toolbar commands
- Menu commands
- Observer calls
- Ruby Tool callbacks
- Timers

Note that no onTransactionCommit is sent out at this point because the operation wasn't ended correctly.

SU2016 Observer Behaviour

What might break your extension

We have worked hard to minimize breaking changes to the API, but some were inevitable in order to make them safe from crashing, memory corruption and interfering with the undo stack. Breaking changes should mainly manifest itself in code which isn't correctly handling operations and observer events.

However, there might be subtle changes that could break seemingly harmless code which didn't validate the observer parameters.

Not Error Safe

```ruby
module Example

  class ExampleLayerObserver < Sketchup::EntityObserver
    def onLayerAdded(layers, layer)
      if layer.visible? # <-- Layer might be deleted!
        # Do something...
      end
    end
  end # class

end # module
```

Since observer events are not queued there's a higher risk of getting deleted entities if the operation create temporary entities.

However, there has always been the possibility that another observer remove the entity before your own observer is triggered.

Error Safe

```ruby
module Example

  class ExampleLayerObserver < Sketchup::EntityObserver
    def onLayerAdded(layers, layer)
      return if layer.deleted? # <-- Always check the validity of entities!
      if layer.visible?
        # Do something...
      end
    end
  end # class

end # module
```

All arguments from an observer events should be validated before acted upon as there is always the chance the data has expired.

Even if you created the entity there might be observers from other extensions that modify it before your own observer trigger.

SU2016 Observer Behaviour

Dummy events from Observer base classes removed

If you inspect the methods defined for the Observer base classes, such as Sketchup::EntityObserver, Sketchup::LayersObserver you will find that in SU2016 they are gone. This was done in order to optimize the way we send notifications. When the dummy methods are not predefined we are able to determine which events are actually listened to by the API user and which is ignored. This allows us to optimize the way we record and dispatch notifications.

Based on our evaluation we could not find any extensions that relied on the dummy methods being there. However, should this have an impact on you, please report back to the team describing your use case.

SU2015

```ruby
class Sketchup::EntitiesObserver

  def onActiveSectionPlaneChanged(entities)
  end

  def onElementAdded(entities, entity)
  end

  def onElementModified(entities, entity)
  end

  def onElementRemoved(entities, entity_id)
  end

  def onEraseEntities(entities)
  end

end # class
```

SU2016

```ruby
class Sketchup::EntitiesObserver

end # class
```

SU2016 Observer Behaviour

Known Issues (SU2016 Alpha 1)

- Zombie Sketchup::Axes
- Zombie Sketchup::Page
- Zombie Sketchup::Material

Busting the Zombie Entity Apocalypse

In previous versions of SketchUp you might get a new Ruby object for entities that has been deleted and they would not always be marked as deleted. This was a source to crashes and memory corruption that could result in odd bugs like Sketchup::Face.edges returning an array that didn't contain only Sketchup::Edge objects.

This should now we fixed. If you observe such behaviour in SU2016 and beyond please report the issue back to the SketchUp team.

Example

```ruby
module Example

  class ExampleEntityObserver < Sketchup::EntityObserver
    def onEraseEntity(entity)
      puts "onEraseEntity(#{entity})"
      puts "> Deleted: #{entity.deleted?}"
    end
  end # class

  # Example.zombie_entities
  def self.zombie_entities
    puts "SketchUp version: #{Sketchup.version.to_i}"
    model = Sketchup.active_model
    edge = model.active_entities.add_line([0, 0, 0], [9, 9, 9])
    observer = ExampleEntityObserver.new
    edge.add_observer(observer)
    puts "Removing entity: #{edge}"
    edge.erase!
  ensure
    edge.remove_observer(observer) if edge.valid?
  end

end # module
```

SketchUp 2015 Result

```
SketchUp version: 15
Removing entity: #<Sketchup::Edge:0x0000000b5574f8>
onEraseEntity(#<Sketchup::Edge:0x0000000b5571b0>)
> Deleted: false
```

Notice the observer returns a new Ruby object which isn't marked as deleted.

SketchUp 2016 Result

```
SketchUp version: 16
Removing entity: #<Sketchup::Edge:0x0000000a4fcdb0>
onEraseEntity(#<Deleted Entity:0xa4fcdb0>)
> Deleted: true
```

The observer now correctly returns the correct Ruby object marked as deleted.

Operations — Best Practices

Rule 1: *One* user action should produce only *one* undo action

Being able to undo is a critical part of the user experience. A user should be able to safely use any tool or function and easily undo the action in one step. Without this the user risk losing their work since last save if the operation didn't produce the desired result and flooded the undo stack.

Bad Behaviour

```ruby
module Example

  def self.operation_bad_behaviour
    model = Sketchup.active_model
    entities = model.active_entities
    # This will create 10 items on the undo stack, one per face and one per
    # attribute. This makes it hard for users to revert action that modifies
    # the model.
    5.times { |i|
      face = entities.add_face([0, 0, i], [0, 9, i], [9, 9, i], [9, 0, i])
      face.set_attribute('Example', 'Hello', "World #{i}")
    }
  end

end # module
```

Beware that setting attributes are also undoable actions.

Recommended Pattern

```ruby
module Example

  def self.operation_best_practice
    model = Sketchup.active_model
    entities = model.active_entities
    model.start_operation('Hello World', true)
    begin
      5.times { |i|
        face = entities.add_face([0, 0, i], [0, 9, i], [9, 9, i], [9, 0, i])
        face.set_attribute('Example', 'Hello', "World #{i}")
      }
    rescue
      model.abort_operation
      raise
    end
    model.commit_operation
  end

end # module
```

Note the error handling that aborts the operation upon errors.

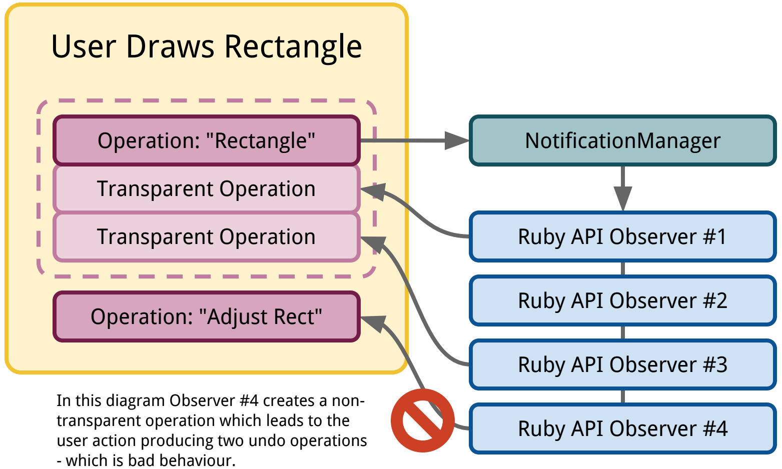
Operations — Best Practices

Rule 2: Observers should create transparent operations

Making changes to the model based on observer events is challenging.

Prior to SU2016 one should never do model changes directly in the observer callback as that could crash SketchUp or disrupt the undo stack.
For these older versions one should follow the pattern outlined in this GitHub repository: https://github.com/SketchUp/sketchup-safe-observer-events

In SU2016 this is now possible, but one must make sure to make the operation transparent so that it doesn't add extra undo steps for the user.



In this diagram Observer #4 creates a non-transparent operation which leads to the user action producing two undo operations - which is bad behaviour.

Recommended Pattern

```
module Example

  # Note: This example is valid for SU2016+ only! Older versions needs a
  # different pattern: https://github.com/SketchUp/sketchup-safe-observer-events
  class ExampleLayerObserver < Sketchup::EntityObserver
    def onLayerAdded(layers, layer)
      # Make sure the entity is valid.
      return if layer.deleted?
      model = layer.model
      # When starting a new operation the first argument will not be visible
      # to the user.
      # Note: Never user the third argument! It's notoriously difficult to use
      # as one cannot be sure what is the next operation.
      # The important argument here, for starting operations in observer events
      # is the fourth one - making the operation transparent to the previous.
      model.start_operation('Hello World', true, false, true)
      begin
        layer.set_attribute('Example', 'Hello', "World")
      rescue
        # Note: Never abort a transparent operation, it will abort the operation
        # it chains to. Instead, try to clean up and simply commit in order to
        # make sure the operation is closed.
        model.commit_operation
        raise
      end
      model.commit_operation
    end
  end # class

end # module
```